# Java Persistence API

## What is the Java Persistence API (JPA)?

The Java Persistence API (JPA) is a standard API for accessing databases from within Java applications.

The main advantage of JPA over JDBC (the older Java API for interacting with databases) is that in JPA data is represented by classes and objects rather than by tables and records as in JDBC. Using plain old Java objects (POJO) to represent persistent data can significantly simplify database programming.

A JPA implementation (sometimes referred to as a JPA provider) is needed in order to interact with a relational database such as Oracle, DB2, SQL Server or MySQL. The popular JPA implementations are Hibernate, TopLink, EclipseLink, Open JPA and DataNucleus. These implementations are Object Relational Mapping (ORM) tools. The mapping bridges between the data representation in the relational database (as tables and records) and the representation in the Java application (as classes and objects).

Mapping Java objects to database tables and vice versa is called *Object-relational mapping* (ORM). The Java Persistence API (JPA) is one possible approach to ORM. Via JPA the developer can map, store, update and retrieve data from relational databases to Java objects and vice versa. JPA can be used in Java-EE and Java-SE applications.

ObjectDB is the only database management system with built in support of the Java Persistence API (JPA). By interacting with ObjectDB using standard JPA you can keep your application portable. The unique benefit of using ObjectDB is that the overhead of an intermediate ORM layer is eliminated. That simplifies and accelerates the development process and makes your application run much faster.

JPA permits the developer to work directly with objects rather than with SQL statements. The JPA implementation is typically called persistence provider.

The mapping between Java objects and database tables is defined via persistence metadata. The JPA provider will use the persistence metadata information to perform the correct database operations.

JPA metadata is typically defined via annotations in the Java class. Alternatively, the metadata can be defined via XML or a combination of both. A XML configuration overwrites the annotations.

The following description is based on the usage of annotations.

JPA defines a SQL-like Query language for static and dynamic queries.

Most JPA persistence providers offer the option to create the database schema automatically based on the metadata.

- Entity

  A class which should be persisted in a database it must be annotated with javax.persistence.Entity. Such a class is called Entity. JPA uses a database table for every entity. Persisted instances of the class will be represented as one row in the table.

All entity classes must define a primary key, must have a non-arg constructor and or not allowed to be final. Keys can be a single field or a combination of fields.

JPA allows to auto-generate the primary key in the database via the @GeneratedValue annotation.

By default, the table name corresponds to the class name. You can change this with the addition to the annotation @Table(name="NEWTABLENAME").

- Persistence of fields

The fields of the Entity will be saved in the database. JPA can use either your instance variables (fields) or the corresponding getters and setters to access the fields. You are not allowed to mix both methods. If you want to use the setter and getter methods the Java class must follow the Java Bean naming conventions. JPA persists per default all fields of an Entity, if fields should not be saved they must be marked with @Transient.

By default each field is mapped to a column with the name of the field. You can change the default name via @Column (name="newColumnName").

- Relationship Mapping

JPA allows to define relationships between classes, e.g. it can be defined that a class is part of another class (containment). Classes can have one to one, one to many, many to one, and many to many relationships with other classes.

A relationship can be bidirectional or unidirectional, e.g. in a bidirectional relationship both classes store a reference to each other while in an unidirectional case only one class has a reference to the other class. Within a bidirectional relationship you need to specify the owning side of this relationship in the other class with the attribute "mappedBy", e.g. @ManyToMany(mappedBy="attributeOfTheOwningClass".

Relationship annotations:

- @OneToOne
- @OneToMany
- @ManyToOne
- @ManyToMany